

VASTU : An Environment for Distributed Object Oriented Programming

by
E. DHANABAL



Th
CSE/1998/4
D535V

Department of Computer Science & Engineering
INDIAN INSTITUTE ON TECHNOLOGY KANPUR
May 1998

SE
998
M
HA
AS

VASTU: An Environment for Distributed Object Oriented Programming

*A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology*

*by
E. Dhanabal*

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
April, 1998

20 MAY 1998 / CSE

CENTRAL LIBRARY

Inv No A 125498

CSE-1998-M-DHA-VAS

Entered in system _____

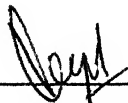
On
29-6-98



A125498

Certificate

Certified that the work contained in the thesis entitled "*VASTU: An Environment for Distributed Object Oriented Programming*", by *E. Dhanabal*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



(Dr. Deepak Gupta)

Assistant Professor,

Department of Computer Science & Engineering,

Indian Institute of Technology,

Kanpur.

April, 1998

Abstract

Designing a programming model for distributed systems that is not too different from the programming model for sequential systems is a highly desired and challenging task. Of the programming languages for sequential systems, object oriented languages lend themselves rather easily to distributed extensions. In this thesis, the design and implementation of a distributed object oriented programming environment(VASTU) is described. VASTU is based on the object oriented programming language *C++*. Using the VASTU system, programmers can transparently create, access and destroy remote active objects. Active classes are specified using the VASTU ADL(Active Class Description Language). The compiler for this language generates *C++* code that hides the details of distribution and provides a transparent interface to the clients. An object server runs on every machine in the system and is responsible for creating objects on that machine. Access to active objects is controlled using capabilities. The VASTU system itself is used to implement a simple name server that can be used to name objects.

Acknowledgments

I express my sincere gratitude to my thesis supervisor Dr. Deepak Gupta, for his valuable guidance and motivation that he gave me throughout this work. He is the one who introduced me to this problem and provided me with lots of ideas. The time that I spent with him is the most crucial and potential learning period of my life. I would like to thank Dr. S. K. Aggarwal for his guidance and help in the compiler part of my implementation. I express my sincere thanks to my thesis examiners for their critical evaluation of this work and their suggestions. I express my sincere thanks to Rajesh, Sakthi, Gomathi, Senthana, Venkat, Kataru, Yugandhar and Raja for helping and motivating me during my work. My cordial thanks to all my classmates(mtech96) and my friends at the Hall IV billiards room who made my stay at the institute a pleasant and memorable one. I express my sincere thanks to my family members for encouraging me to do post graduate studies. Last, but not the least, I would like to thank the CSE department lab staff for providing the facilities and helping me whenever I needed.

Contents

List of Figures	iv
1 Introduction	1
1.1 Programming Models for Distributed Systems	1
1.1.1 Parallelizing Compilers	2
1.1.2 Languages with Logically Shared Address Spaces	2
1.1.3 Languages with Distributed Address Spaces	3
1.2 Object Oriented Programming	4
1.3 Distributed Object Oriented Programming	5
1.3.1 Object Composition	5
1.3.2 Locating the Object	7
1.3.3 Security	8
1.3.4 Object Scheduling	8
1.3.5 Action Management	9
1.3.6 Synchronisation	9
1.3.7 Reliability	10
1.3.8 Invocation Failures	10
1.3.9 Object Mobility	10
1.4 An Overview of Some Existing Systems	11

1.4.1	Emerald	11
1.4.2	Sloop	11
1.4.3	Argus	12
1.4.4	Eden	13
1.4.5	Aeolus	14
1.4.6	CORBA	15
1.5	Scope of the Present Work	15
1.6	Organization of the Report	16
2	System Design	17
2.1	Design Goals	17
2.2	Programming Language Considerations	18
2.3	Nature of Objects	20
2.4	Active Class Description Language	21
2.5	Active Object Creation	22
2.6	Naming of Objects	23
2.7	Concurrency within Objects	23
3	Implementation	25
3.1	Active Class Description Language	25
3.2	Object Handles	29
3.2.1	Authentication	30
3.3	Object Creation	32
3.4	Invoking Remote Objects	34
3.5	Destroying Active Objects	35
3.6	Inheritance	36
3.6.1	Method Dispatching	38

3.7	Name Server	39
3.8	A typical Execution Scenario	40
4	An Example Application	43
4.1	Description of the Example	43
4.2	The ADL Specification of the Print server class	44
4.3	Generating Code from the ADL Specification	45
4.4	Building the Print Server from the Generated Code	45
5	Conclusion	48
	Bibliography	50
A	Active Class Description Language Grammar	53

List of Figures

1	Passive object model	6
2	Active object model	6
3	Using the ADL compiler	26
4	active_proxy_root class declaration	27
5	active_root class declaration	28
6	HANDLE structure	29
7	OBJ_CR_PARAMS structure	32
8	Destruction of an active object	36
9	A simple inheritance hierarchy	37
10	An execution scenario	42
11	ADL specification of the print server class	44
12	Modification of the generated main method	46
13	Synchronisation between the schedule method and the status method	47

Chapter 1

Introduction

The development of low cost powerful microprocessors and high speed computer networks have revolutionized the way computing is done. It was these two developments that caused the advent of distributed systems. A distributed system can be defined as a collection of computers or microprocessors interconnected by a communication network, where the computers communicate with each other through message passing and not by sharing memory. The key difference between a distributed system and a network of computers is the transparency provided by the distributed system. The system hides the presence of multiple computers and provides a single system view to the user.

Distributed systems have many advantages to offer over computing with sequential machines. The first one is that programs can exploit the parallelism available due to multiple CPUs. The second is higher reliability which comes about due to the ability of the system to tolerate partial failures. Another advantage is that such systems are well suited for incremental growth.

1.1 Programming Models for Distributed Systems

The challenge in distributed computing is to provide a model for programming which allows the programmer to achieve the above goals, yet it is not too different from the way sequential machines are programmed. There are many ways in which this can

be achieved.

1.1.1 Parallelizing Compilers

Some compilers, for example, [14, 21, 28] can extract parallelism from a sequential program and thus make it suitable for efficient execution on multiple CPUs. However, these are best at extracting fine grained parallelism which, due to the relatively large communication delays, is unsuitable for a collection of autonomous machines connected by a communication network. Moreover, even the best of such compilers can extract only a limited amount of parallelism from the program. Also, using such a model of programming, problems which are inherently distributed in nature can not be effectively modelled. This model is very efficient only when the parallelism is made explicitly visible to the programmer.

1.1.2 Languages with Logically Shared Address Spaces

In these languages, parallel units of a program have a logically shared address space and communicate through data stored in the shared address space. Note that this does not mean that physical shared memory is needed for the implementation of such languages. These languages can be further classified into three categories.

- 1 Functional languages
- 2 Logic languages
- 3 Distributed data structures

The implicit parallelism in functional languages is especially suited for closely coupled architectures like data flow machines. But, they are unsuitable for distributed systems which are mostly coarse grained in general. However, there exist some languages like *ParAlf*[15] which provide a mapping notation for efficiently distributing computation among multiple processors. There exist parallel logic languages like *PARLOG*[7] and languages based on distributed data structures like *Linda*[11]. But, most of these

languages are designed for multiprocessor machines with shared memory and would be relatively inefficient in a distributed environment.

1.1.3 Languages with Distributed Address Spaces

In these languages, the parallel threads of computations within a program communicate by sending messages to each other. The address spaces of different computations do not overlap, so the address space of the whole program is distributed.

The most basic model in this category is a group of sequential processes running in parallel and communicating through synchronous message passing. Example of a language based on this model is *Occam*. Some languages use asynchronous message passing to increase the parallelism, i.e., the caller does not wait for the messages to be delivered but immediately continues to do other computation.

Both of the above models make the underlying hardware explicitly visible to the programmer and that is what makes the life of the programmer, who is accustomed to programming on uni-processor machines, difficult. What one would like to have is a model which is as close as possible to the way sequential programming is done. *Remote Procedure Call* is a step in this direction. Using this mechanism, the interaction between the two processes appears as a normal procedure call to the programmer. This idea was first proposed by Birrell and Nelson in [5]. There are several implementations of this idea such as Sun-RPC[24], Apollo RPC[20] etc.

Object oriented languages lend themselves rather easily to distributed extensions. All one has to do is to think of objects as independent entities just like processes in the above models. Performing an operation on an object can be visualised as sending a message to the object. Parallelism can be achieved by making the objects *active*. That is, they just do not wait for requests, they do some operation on their own also. More parallelism can be achieved by making the object multi-threaded and/or making use of asynchronous method invocations.

1.2 Object Oriented Programming

Approaches to programming have changed dramatically since the invention of computers. Initially, programmers had to directly code in machine language. Now, high level languages are available which make programming easier. Paradigms of programming have also changed considerably with the increasing complexity of programs. Initially, there was *Structured programming* - the model encouraged by languages like C and Pascal. But, software maintenance became a problem with this approach. Then came the *Object Oriented model* which took the advantages of structured programming and reduced the problem with software maintenance. This model simplified the task of programming since there is a close correspondence between the way a programmer visualized the solution and the actual program.

An *object* is an entity that encapsulates some private *state* information or data and a set of associated *methods* or procedures that manipulate the data. The state of an object is completely protected and hidden from other objects. The only way it can be examined or modified is by the invocation of one of the associated methods. This creates a well-defined interface for each object, enabling the specification of an object's operations to be made public while keeping the implementation of the operations and the representation of the state private.

A *class* is a template from which objects can be created. All objects that are created using this template are said to be instances of the class.

Inheritance is a mechanism by which new classes can use code of existing classes, simply by specifying how the new classes differ from the existing ones. In such a situation, the old class is called as the *base class* or the *super class* and the new class is known as the *derived class* or the *sub class* and the relationship between them is termed as an *IS-A* relationship. The derived class gets all the functionality of its super class. It can also add its own functionality or override some of the functionality of the super class.

A programming language is termed as *object oriented* if it supports the notions of classes and inheritance. By this definition, languages such as C++[22] and Smalltalk[12]

are object oriented. An object oriented program has many objects which remain passive and respond to method invocations whenever requested. There is only one thread of control at any given point of time.

1.3 Distributed Object Oriented Programming

A distributed object oriented programming language provides the features of an object oriented programming language as well as supports a distributed computing environment. A distributed object oriented program typically consists of many objects each of which may be executing on a different machine and communicating with each other through some communication protocol. In the following sections, we briefly look at some of the issues in supporting distributed object oriented programming[6].

1.3.1 Object Composition

The relationship between processes and objects characterises the composition of the objects. There are two approaches to this.

First one is the *Passive object model*. In this model, processes and objects are completely separate entities. A process is not bound nor it is restricted to a single object.

When a process makes an invocation on another object, its execution in the current object is temporarily suspended. Conceptually, the process is then mapped into the address space of the second object, where it executes the appropriate operation. When, it completes this operation, it is returned to the first object and it resumes the original operation. A typical scenario is depicted in Figure 1, where P is the process which invokes a method of object A.

Objects are passive entities here. They do not do anything on their own. They just respond to method invocation requests.

Second model of object composition is the *Active object model*. In this model, objects are active entities. They do something on their own, in addition to servicing method invocation requests. When an object is created, many server processes(or

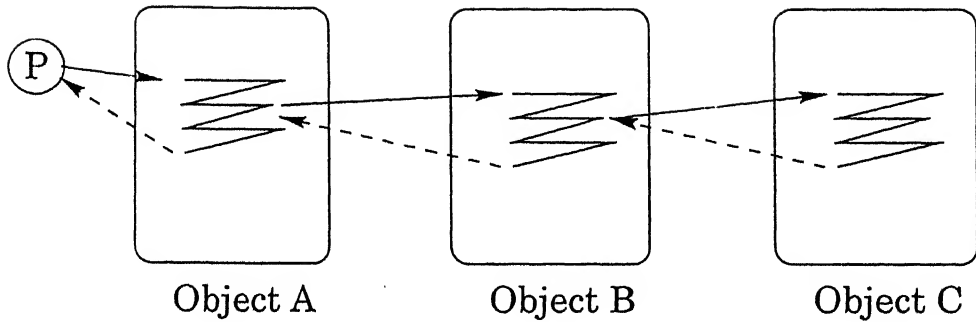


Figure 1: Passive object model

threads) are created and assigned to the object. These processes die when the object is destroyed.

When a client makes an invocation, it forms a request message with the arguments and sends it to the object. A server process of the object accepts this request and performs the specified operation. Upon completion, it forms a reply message with the result of the operation and sends it back to the client object. A typical scenario is depicted in Figure 2, where PA, PB and PC are the server processes attached to the objects Object A, Object B and Object C respectively.

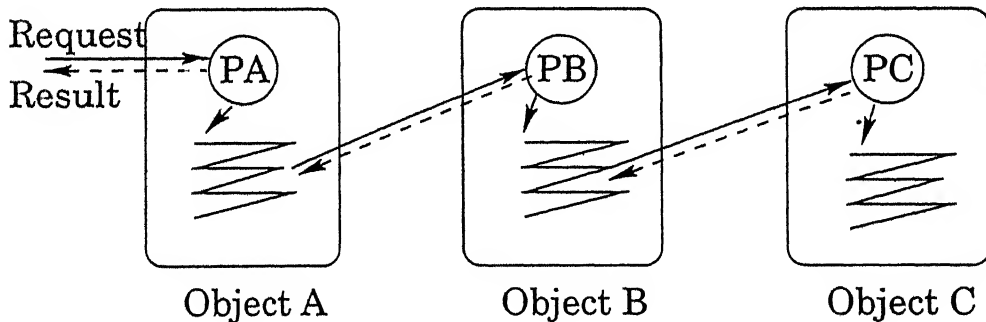


Figure 2: Active object model

If the number of server processes is fixed, then the model is a *static variation* of the active object model. Here, requests that arrive when all servers are busy are queued and serviced later. If a server process is created to handle each incoming

request, the model is a *dynamic variation* of the active object model. The additional overhead in terms of process creation and destruction can be minimised by maintaining a pool of idle processes.

1.3.2 Locating the Object

In a distributed environment, objects can reside on any machine. To invoke a method on an object, the location of the object needs to be known. Therefore, there should be a way by which we can get the location of the object from the name of the object. There are many schemes by which this can be achieved.

One simple scheme is to encode the location of the object within its name. Whenever an invocation is made, the location can be found out just by looking at the name of the object. This is the simplest and the most straightforward scheme. But the disadvantage is that it is not location and migration transparent.

The second approach is to use a *name server*. There is an entity called *name server* running in the system at a well-known location. This maintains the mappings of locations and names of objects in the system. Objects can provide mappings to the name server and can get location of an object from the name server by giving the name of the object. The drawback of this scheme is the centralised nature of the name server. The name server becomes a performance bottleneck as well as a critical point of failure. However, these problems can be addressed by replicating the name server as in DCE[4].

A small cache can be maintained on each machine that records the last known locations of a number of recently referenced remote objects. When a client makes a remote invocation, the cache can be examined to determine if it has an entry for the remote object. If a location is found, the invocation is sent to that machine. If the location can not be found out from the cache, a message is broadcast throughout the network requesting the current location of the object. Every machine that receives the request does an internal search for the object. If it is found, a reply message is returned and the cache on that machine is updated upon receiving the reply.

Forward location pointers can be used to enhance most location schemes. A forward location pointer is a reference used to indicate the new location of an object which has moved from one machine to another. Whenever an object moves from one machine to another, it leaves a forward location pointer at the original machine. To locate an object that has moved, the system can simply follow the chain of pointers. This scheme can not completely handle the problem of finding migrating objects, since some pointers in the chain may be lost due to machine failures.

1.3.3 Security

Providing a security scheme to prevent unauthorized clients from successfully invoking methods of an object is an important issue in a distributed system. There are basically two approaches to solve this problem.

The *Capability* scheme incorporates protection into the naming scheme. A capability works as an address for an object. It is also used to guarantee accesses. Upon creation of an object, clients get back a capability for the object. They have to use the capability for later method invocations on that object. Capabilities are transferable, restrictable and unforgeable.

In the *Control procedure* scheme, every object has a control procedure through which all incoming invocation requests must first pass. This checks the authorization of clients making a request and terminates all invalid requests. It is flexible, since it can support any type of security scheme. This permits each object to provide security tailored to the requirements of the object. If security is unimportant to an object, a minimal security scheme can be used. If an object is security sensitive, a more rigid scheme can be used. This scheme can also be used in addition to the capability based scheme.

1.3.4 Object Scheduling

Object scheduling is the issue of choosing a processor in the system on which a new object should be created. This can be either *explicit* or *implicit*. In an explicit scheduling scheme, the user is responsible for specifying the processor to which an object is

to be assigned. In the implicit scheme, the system is responsible for determining the location of the object. Typically, the system determines the load on the available machines and chooses the one with the lightest load. One drawback of this approach is that, getting accurate load information is very difficult and therefore, a machine which is observed to be lightly loaded may get assigned a number of objects and may become overloaded by the time the object is created on that machine.

1.3.5 Action Management

Action in a distributed object oriented system is a chain of related method invocations. Since actions in a distributed object oriented system execute concurrently, the properties of an action should be maintained. The properties of an action are:

- *Serializability*: Multiple concurrently executing actions should be scheduled in such a way that the overall effect is the same as if they were executed in some serial order.
- *Atomicity*: An action either successfully completes or has no effect.
- *Permanence*: The effects of a successful action or not lost.

To ensure these properties, protocols such as *two-phase commit* protocol [13] must be used.

1.3.6 Synchronisation

Another important function of a distributed object oriented programming system is to ensure that multiple actions that invoke the same object do not conflict or interfere with each other. Failure of this can lead to *cascading aborts*, i.e., an action that has observed the partially modified state of an object resulting from an action that later aborts, also has to be aborted. Two popular synchronisation schemes exist to avoid this. One takes appropriate steps to prevent conflicts from occurring. An action that invokes an object is temporarily suspended if it will interfere with another currently executing action. This scheme is called as *pessimistic* scheme. Another

scheme does not takes steps to prevent conflicts from occurring. Instead, before an action can commit, a check for serializability is done to ensure that the information it has observed is not out-of-date. This scheme is called as *optimistic* scheme.

1.3.7 Reliability

There are two methods to provide object reliability. One is to *recover* a failed object as quickly as possible, limiting the amount of time for which it remains unavailable. Another is to *replicate* objects at multiple machines so that even if one of them fails, the other can continue to service requests.

1.3.8 Invocation Failures

An invocation can fail because of many reasons. It can be because of a client failure or a server failure or a network failure. There are many failure detection algorithms[18] using timeouts and object probes that can detect this.

1.3.9 Object Mobility

An object migration scheme permits an object to move from one machine to another. There are basically two reasons why an object should be migrated: first is to improve performance and the second is to improve availability. Migrating a process in a conventional distributed system is a difficult task. This is due to the problem in moving system dependent information such as values of running clocks, the logical communication paths, etc. In a distributed object oriented programming system, this is simplified, since an object clearly defines the entities that must be moved as a single unit. In addition, the property of location transparency permits a system to determine the new location automatically.

1.4 An Overview of Some Existing Systems

1.4.1 Emerald

Emerald[19] is a distributed object based programming language and run-time system developed at the University of Washington. In Emerald, all entities are objects. Both passive objects and active objects are supported. Concurrency within an object is also supported. A pessimistic synchronisation scheme that uses monitors and an immutable object replication scheme are also implemented. Emerald uses a cache scheme that uses forward location pointers for locating remote objects. It supports object migration wherein processes executing in the object are stopped, the state of the object is sent to the destination machine, the object is created on that machine with the state and the processes are resumed. The system does not take care of requests that arrive while the object is in transition. In Emerald, a single common address space is shared by all objects on a machine. This reduces the cost of creation of objects and enables objects to examine and modify each other directly. But, there are no hardware protections between objects and hence, an object failure can potentially lead to an entire machine failure. Emerald does not support a security scheme and it replicates immutable objects only. There is no notion of classes and inheritance. Only call-by-reference is supported. To reduce the overhead that will arise because of the remote invocations on the objects passed as parameters, the object to be passed is moved physically to the called machine for the duration of the call. This mechanism is termed as *call-by-move* in Emerald. The distributed programs developed in a system and a language like Emerald typically operate in a separate environment and it is more difficult to integrate them into existing systems.

1.4.2 Sloop

Sloop[17] is a parallel language and an environment employing object oriented model. It supports the notion of classes and inheritance. Sloop employs passive object model. It hides the underlying multiprocessor environment with the abstraction of *virtual object space* which consists of a collection of coordinating objects called *domains*. Each domain provides operations for creating objects, getting object location from

name, making a group of objects reside on a specified processor and for replicating an object. Objects interact with domain objects as they interact with other normal objects.

Object types have a feature called exclusion level which can be either *re-entrant* or *indivisible* to indicate whether object's operations can proceed concurrently or not. Even in re-entrant objects, true concurrency is not provided. Once invoked, a re-entrant operation will run until it either surrenders control or invokes another operation. Moreover, programmer should ensure that the re-entrant operations leave the object in a consistent state before surrendering control or invoking some other operation.

For synchronisation, a construct called *await* is provided. Any operation executing the statement *await P* will wait till the condition P becomes true. Sloop does dynamic load balancing by monitoring the load and migrating objects. It uses coroutines rather than processes for operation execution. It also provides real-time support. The user can specify an object, an operation and a time interval. The system will periodically invoke the operation of the object. Both implicit and explicit object scheduling are supported. Object migration can be done both by the user and the system. Naming scheme is based on capabilities. There is no security scheme and the system does not support atomic actions and reliability.

1.4.3 Argus

Argus[16] is a system and a language developed at Massachusetts Institute of Technology. It provides support for fault-tolerant distributed programming. It supports dynamic variation of the active object model and it minimises the overhead in process creation and destruction by maintaining a pool of idle processes. In Argus, objects are called as *guardians* and there is a section called *recover* in a guardian that is executed whenever the object is recovered after a crash. Argus supports transactions and a pessimistic scheme that uses read/write locks for synchronisation. A guardian manager is executed on each machine to create new guardians on it, to detect failures and to recover guardians after a failure. Security and object replication are not supported. Location of the guardian is encoded within its name and hence guardians are

immobile. Clients use an invocation probe scheme to detect failures and servers use a status report scheme to detect invocation failures. To support recovery, a commit log is maintained in secondary storage. Argus does not free the programmer from worrying about the details of concurrency. The programmer must think about deadlocks and starvation and implement the code to avoid them when possible. Moreover, there is no notion of classes and inheritance in Argus.

1.4.4 Eden

Eden[1] is a system and a programming language developed to build distributed object based applications. Eden supports the static variation of the active object model. Each Eden object or Eject has a long-term state that contains the persistent information, a short-term state that contains the volatile information and a code segment that contains the operations. Each Eject has a dispatcher process for accepting requests and assigning to idle server processes, one or more server processes and some maintenance processes. Eden's kernel itself is implemented as a Unix process and it supports primitives for asynchronous message passing, Eject creation and checkpointing. Even though portability of the system is enhanced by having the Eden kernel as a process there is a high overhead of communication between Ejects and the Eden kernel. Eden does not support automatic garbage collection and atomic actions. Eden has a pre-processor which generates stubs for Eden object types and relieves the programmer from worrying about the communication primitives. The generated stubs provide synchronous communication based on the asynchronous communication provided by the kernel. A pessimistic synchronisation scheme that uses monitors, a capability security scheme and a recovery scheme that uses checkpoints are also provided. In Eden, one Eject can have only one capability. This prevents grouping together of two Ejects into one without also changing their interfaces. Checkpointing in Eden replaces an entire Eject state rather than selectively updating even for very small changes in the state. Object replication is also supported. It supports cache scheme for locating objects. Timeout scheme is used by clients to detect invocation failures. Implicit object scheduling is also supported.

1.4.5 Aeolus

Aeolus[3] is a programming language for implementing distributed fault-tolerant programs on the Clouds[8] distributed operating system. Both Aeolus and Clouds were developed at Georgia Institute of Technology. Aeolus supports only passive objects. An object consists of a number of segments: a code segment, one or more data segments for persistent data and a number of heap segments for volatile data. Code segments can be shared between multiple objects. When an invocation is made on an object, the associated process enters the object through one of the entry points in the code segment, executes the corresponding operation and then leaves the object. A programmer can specify whether automatic recovery is needed or custom recovery is needed. In the former case, entire state of an object will be checkpointed and in the latter case, only those parts of the object that have been indicated by the programmer will be checkpointed. The programmer can also specify whether automatic synchronisation which is of pessimistic nature is needed. The programmer can also achieve synchronisation by using locks explicitly. The drawback of this scheme is that serializability of actions can not be enforced by the system. Even though, Aeolus gives more flexibility for optimising recovery and synchronisation, the many features thus introduced make it a complex language. A capability scheme is supported for security. It also supports a recovery scheme and object replication. When an invocation is made on a remote object, a local communication manager broadcasts a *search and invoke* request containing the capability of the object, name of the operation and the parameters. Each communication manager receiving this request, does an internal search for the object. If found, the communication manager accepts the request, creates a worker process to execute on behalf of the client. When the operation terminates, a reply is sent back to the communication manager on the client machine. Clients use a probing scheme and servers use a timeout scheme for detecting invocation failures. Object scheduling is also supported. Aeolus makes direct use of Clouds primitives and hence it is very difficult to port the programs developed in Aeolus to be run on other operating systems.

1.4.6 CORBA

Heterogeneity present in large networks like the Internet and corporate intranets enables one to use the best combination of software and hardware for each part of the enterprise. But the components developed in different parts cannot be integrated together unless there are right standards for interoperability and portability. In view of this problem, Object Management Group(OMG) was formed to develop, adapt and promote standards for the development and deployment of applications in distributed heterogeneous environment. Common Object Request Broker Architecture(CORBA)[27] is one of the specifications proposed by OMG for the development of interoperable and portable software. There are many commercial implementations of CORBA such as Orbix[2].

CORBA supports both passive objects and active objects. Objects are named using an approach similar to capability scheme. The capability is called as object reference in CORBA. Security and naming service are incorporated as CORBA services which can be used by clients. Both synchronous calling mechanism and asynchronous calling mechanism are supported. CORBA hides the location of the object, the execution state of the object and the language in which the object was created from the client. So, a client written in one language can invoke methods of an object written in another language possibly executing on another machine running a different operating system. A locking mechanism is provided by the concurrency service of CORBA for synchronising concurrent callers of an object. Atomic actions and recovery are supported by the transaction service of CORBA. CORBA has no client operations for object creation. Objects are created by invoking methods on built-in objects called factory objects. CORBA does not allow objects to be passed by value and it does not replicate objects also.

1.5 Scope of the Present Work

All the above systems either support each feature of a distributed object oriented system in a rigid manner or do not support it at all. CORBA does not allow the clients to create objects in a transparent manner, i.e., the client has to invoke methods

of some factory objects to create other objects. Emerald, Argus and Eden do not support classes and inheritance. Emerald, Aeolus and Argus are languages developed especially for specific systems and hence programs written using those languages are not portable. Emerald, Sloop and Argus do not support a security scheme. CORBA and Eden provide a security scheme that is inflexible. Argus and Eden support active objects only whereas Sloop and Aeolus support passive objects only. Argus and CORBA do not replicate objects. Atomic actions and reliability are not supported in Sloop. Object migration is not supported in Argus.

The scope of this work is to design and implement a distributed object oriented programming environment(VASTU) that would enable programmers to easily write distributed object oriented applications. VASTU supports classes and inheritance. It supports both active objects and passive objects. Object creation, usage and destruction are done in a transparent manner in the sense that they are similar for both passive objects and active objects. The security scheme is based on capability and control procedure. This scheme is highly flexible. Naming scheme is based on capability and name server. Internal concurrency within an object is also supported optionally. Clients use a timeout scheme to detect invocation failures. Both implicit and explicit scheduling schemes are supported. VASTU makes use of an existing object oriented language(C++) and hence the programs written using VASTU are highly portable. It currently does not support object migration and replication, atomic actions etc.

1.6 Organization of the Report

The rest of the report is organized as follows. The design of VASTU is presented in Chapter 2. In Chapter 3 we describe its implementation. In Chapter 4 we describe how to develop distributed object oriented programs in VASTU with an example. A summary of the work done is presented in Chapter 5.

Chapter 2

System Design

In this chapter, we describe the design of VASTU. First, we present the design goals. Then, we describe the possible design approaches and choose one that best meets our goals. We then proceed to describe the nature of objects in VASTU, the way they are created and the way they are named. Finally, we describe how we support internal concurrency within objects in VASTU.

2.1 Design Goals

Our aim is to create a distributed object oriented programming environment(VASTU) which supports object creation, usage and destruction in a transparent and secure manner, and with reasonable performance. The programs developed in such an environment should be highly portable. Eventually, VASTU should also support features such as persistent objects, replicated objects and object migration. We did not plan to support these features in the first version of the system that is described here. However, in order to add support for such features in the future, the system should be flexible enough to accommodate the necessary changes. We want VASTU to support both the following scenarios — one in which a program creates some objects on remote machines, uses them and then destroys them; and the other in which an object acts as a server and accepts requests from any client program, not only the

one that created it. The first situation would occur when a program wishes to distribute its computations among several processors on the network. Support for the latter scenario would enable the use of the system for the development of the usual client-server kind of applications.

2.2 Programming Language Considerations

A distributed object oriented program, in general, consists of several objects, which may exist at several different machines. Thus when an object invokes a method of another object, the target object may be either local or remote. Method invocation on a remote object needs to be handled differently than method invocation on a local object. In the former case, we have to send a request message to the process implementing the remote object. But in the interest of transparency, the programmer should be able to make the two kinds of invocations in the same manner. Thus, the run-time system should know which objects are remote and which are local. At the time of creation of an object, the run-time system needs to know whether the object should be created locally or remotely. Since the system cannot, in general, make this decision intelligently, the programmer needs to specify this in some way. Thus, it would seem that for supporting distributed object oriented programming either a new language is required or extensions need to be made to some existing object oriented language. *Emerald*[19], *Sloop*[17], *Argus*[16], and *Acolus*[3] are examples of object oriented languages explicitly designed for supporting distributed programming.

However, another way to meet the requirements outlined above is to not use a new language or extend an existing one but to use automatic code generation. Similar kind of problems are also encountered in providing support for distributed programming using RPC. In the case of RPC also, we have local procedures and remote procedures and calling a remote procedure has to be handled differently than calling a local procedure. But, to a programmer, both should appear the same. The way this problem is solved in most RPC systems is as follows. The programmer specifies the declaration of remote procedures in a separate specification language. The compiler for this language generates client and server “stub” procedures in a native language

like C. The programmer compiles the remote procedures along with the server stub in a separate binary executable file and executes it on a remote machine. The client stub procedures are linked with the client program. Whenever the client calls a remote procedure, it is the corresponding stub procedure that actually gets called. This stub procedure packs parameters into a message, sends it to the server and waits for a reply. Upon getting this message, the server stub sees what procedure has to be called, extract parameters, calls the required procedure and returns the results to the client in a message. The client stub procedure returns the results that it gets from the server to the caller.

This approach is advantageous because there is no need for the users to learn a new language for developing distributed programs and the implementation of the approach is much easier. The specification language is usually syntactically very similar to the native language supported by the system. *CORBA*[27] follows a similar approach to support distributed object oriented programming. We also follow the same technique and define a separate language for specifying the classes whose remote objects will be used in a program. The compiler for this language generates a “client-proxy” class and server side code for dispatching requests etc. We use C++ as the native language in which the code is generated and in which the programmer is required to program. The programmer compiles the server code into a separate binary executable file. He uses the client-proxy class for creating remote objects in his program. Whenever he creates an object of the proxy class, the constructor for the proxy class creates the real remote object by executing the corresponding server using a mechanism that will be described later. Later, when the client invokes a method on the remote object, the method of the proxy object gets called. It packs parameters into a message and sends it to the remote object. Upon receiving this message, the generated code for the remote object determines what method has been invoked, extracts parameters from the message, invokes the required method and returns the results.

2.3 Nature of Objects

Structure of objects in a distributed object oriented system influences its overall design. An object oriented program typically consists of several objects of widely varying granularities. The granularity of an object is determined by its size and the amount of computation it does. On one end, we have *coarse-grain* objects which have large size and do a lot of computation. On the other end, we have *fine-grain* objects which have small size and do a little computation. The time taken for execution by a method of a coarse-grain object is very high when compared with the overhead time in invoking that method. But, in the case of a fine-grain object, the overhead in method invocation is higher. An example for a coarse-grain object would be a database object and an example for a fine-grain object would be a complex number object.

Most of the existing operating systems do not support object abstraction at the operating system level. Therefore, we have to implement objects as processes which are supported by most operating systems. Thus, all the objects of a program have to be implemented using a set of processes. A process in this set can contain either one object or a collection of objects. If both the caller object and the called object belong to the same process, the mechanism of the call can be the same as in a usual sequential object oriented language and the overhead in calling would be relatively less. If the two objects are in different processes, the call would have a much higher overhead because messages have to be exchanged between these processes for passing parameters and returning results. Therefore, in our design, only coarse-grain objects whose method execution time is higher than the calling overhead are implemented as separate processes. Fine-grain objects can be contained within a coarse-grain object as part of the coarse-grain object's state. In this case, their methods can be invoked from within the containing coarse-grain object only. A coarse-grain object which is implemented as a separate process is called an *active* object. Of course not every coarse-grain object in a program need be made an active object.

If an object invokes the method of another object that resides in a different process (which may reside on the same machine or on a different machine), it has to send a message containing the parameters of the method and information about the method

invoked to the called object. Upon receiving the message, the called object needs to see what method is being called, extract parameters from the message, invoke the required method and return the result in the form of a message. In our design, each active object has a *main* method that does this job. As will be discussed later, this method is usually automatically generated. Usually, this method calls a function that executes in an infinite loop, waiting for requests and dispatching them. However, the programmer can modify this default behaviour of the generated *main* method. For example, it can be modified to create another thread that does some other activities in the background. The original thread can proceed to wait for requests and dispatch them as usual. For example, in the case of a printer manager object that controls the job queue of a printer, scheduling the job queue can be the background activity.

2.4 Active Class Description Language

As already discussed that we use a separate language for describing the active classes, i.e., those classes whose objects will be active. This language is called *Active class Description Language* (ADL). We have implemented an ADL compiler which when given the declaration of an active class generates a *client proxy* class, eXternal Data Representation(XDR) routines, *server* class and server code. XDR[25] is a standard for representation of data exchanged between two communicating entities on two different machines, to take care of the architectural differences between the two machines. XDR routines are needed to convert data to and from the XDR format.

The user can use the client proxy class for creating objects of the active class in his program. The creation, usage and destruction of active objects and passive objects are done in the same way. It is the generated proxy class which hides the process creation, remote method invocations and process destruction from the user. When an object of the client proxy class is created, it creates a process implementing the corresponding active object using the mechanism described in the next section and gets a *handle* for the active object. This *handle* acts as a capability for the active object. Later, when a method of the proxy object is invoked, the method code which is generated by the ADL compiler packs the arguments of the method, the handle

and information about the method invoked into a message. The Sun-RPC protocol message format is used for this message. It then sends this message to the process implementing the active object and returns the result that it gets to the caller.

The ADL compiler uses the *rpcgen* program [23] to generate XDR functions for packing and unpacking arguments into messages. Parameters can be any type that is supported by *rpcgen*. These include base types such as integers, characters etc., arrays, structures etc., of arbitrary types. In addition, references to active objects can also be passed. This is done by sending the handle for the active object in the message. Since objects that are not active do not have network wide handles these cannot be passed as parameters of methods.

For the server version of the active class, the ADL compiler generates an *authenticate_handle* method, an *authenticate_request* method, a *dispatch* method and a *main* method. The *authenticate_handle* method and the *authenticate_request* method act as the control procedures for the object. The *dispatch* method accepts and services method invocation requests. As described earlier, the *main* method initializes the object and calls a function which executes an infinite loop, waiting for requests. This function uses the *dispatch* method to dispatch the incoming requests. The programmer can use his own versions of these methods if he chooses not to use the ones generated by the ADL compiler.

2.5 Active Object Creation

A client proxy object can not create a process implementing the corresponding active object on a remote machine by itself. An *object server* is used to solve this problem. An object server runs on every machine in the system and is responsible for creation of *active* objects on that machine. It maintains a *repository* of binary executables corresponding to *active* classes. The user compiles the server side code generated for the active class into a separate binary executable file and submits it to this repository.

Whenever a client proxy object is created, it sends a request message containing the name of the active class to the object server on an appropriate machine for creating an active object of that class. Upon receiving the request message, the object server

executes the program corresponding to the specified class and returns a *handle* to the proxy object.

2.6 Naming of Objects

As we saw in the last section, an object that creates an active object gets back a handle for the active object. The object uses the handle for method invocations on the active object. But, in some cases, the creator of an active object and its user may be two different entities. In such a situation, the user does not know the handle of the active object whose methods he wants to invoke. A name server can be used to solve this problem. An active object can register its name and handle with a name server optionally. Any object which knows the name of an active object can obtain the handle of the active object by contacting the name server with the name. A name server runs on each machine in the system and maintains a table of <object name,handle> mappings for objects which are present on the local machine. It provides services to add a mapping, remove a mapping and get an object's handle given its name.

Currently we have implemented only a very simple name server that manages the names of objects on the local machine. It implements a flat name space. This is sufficient for the first version of the system that is described here. However, the name space can be made more complex and the name service can be made distributed in nature later on.

2.7 Concurrency within Objects

VASTU allows active objects to be multi-threaded. Programmers can use POSIX thread facilities[9] to create threads. All our library functions are either thread-safe or have both thread safe and unsafe versions. The ADL compiler has an optional switch. If this switch is used, the generated code creates a new thread for servicing each incoming request. It is the programmers responsibility to ensure synchronisation between these threads by using POSIX thread synchronisation facilities such as *mutex*

and *condition variables*. Therefore, a programmer can choose either a single thread sequential dispatching scheme or a multi-threaded concurrent dispatching scheme where a new thread is created for each incoming request. Actually, there can be other dispatching schemes in between these two. For example, a pool of idle threads can be maintained and one of these threads can be assigned to each incoming request. Any such scheme can be easily implemented by the programmer by modifying the generated code of the *dispatch* method. Later, if we find that some such scheme is frequently used, the ADL compiler can be modified so that it can generate code for that scheme also.

Chapter 3

Implementation

In this chapter, we describe the implementation of VASTU. We first describe the Active Class Description Language(ADL), the usage of the ADL compiler and the code generated by the compiler. Then, we proceed to describe the structure of the handle, the authentication mechanism and the way explicit scheduling of objects is supported. Then, we describe the way objects are created, invoked and destroyed and the way inheritance is supported. Finally, the implementation of the name server is presented followed by the description of a typical execution scenario. The underlying system that we have used for our implementation is a collection of Unix machines connected by a communication network. We have used RPC as the communication mechanism and the particular RPC implementation that we have used is Sun-RPC implemented on Unix platforms.

3.1 Active Class Description Language

An ADL specification file contains the declaration of an active class and the data types that it uses. The file is structured into three sections. First section has constructs like *#defines* and *#includes* which need to be passed untouched to the output files generated. In the second section, the programmer writes the declaration of data types of arguments of publicly accessible methods of the active class. The syntax of this section is exactly same as the *XDR*[25] language of the Sun-RPC. The third section

contains the declaration of the active class. The syntax of the class declaration is similar to the syntax of the class declaration in *C++* with the exception that the class cannot have *public* data members, *static* members and *friends*. The grammar of this language is given in Appendix A.

The ADL compiler is implemented using the Unix compiler-compiler tools *lex* and *yacc*. When a file containing the specification of an active class is given as input, this compiler generates a *client-proxy* class, server class, *main* function for the server program and XDR routines. The specification of the data structures given in the second section of the input file is passed through the *rpcgen*[23] compiler to generate the XDR routines.

The programmer compiles his program along with the client-proxy class and the XDR routines to generate the client program. He compiles the *main* function of the server program along with the server class and XDR routines into a separate binary executable file. This is illustrated in Figure 3.

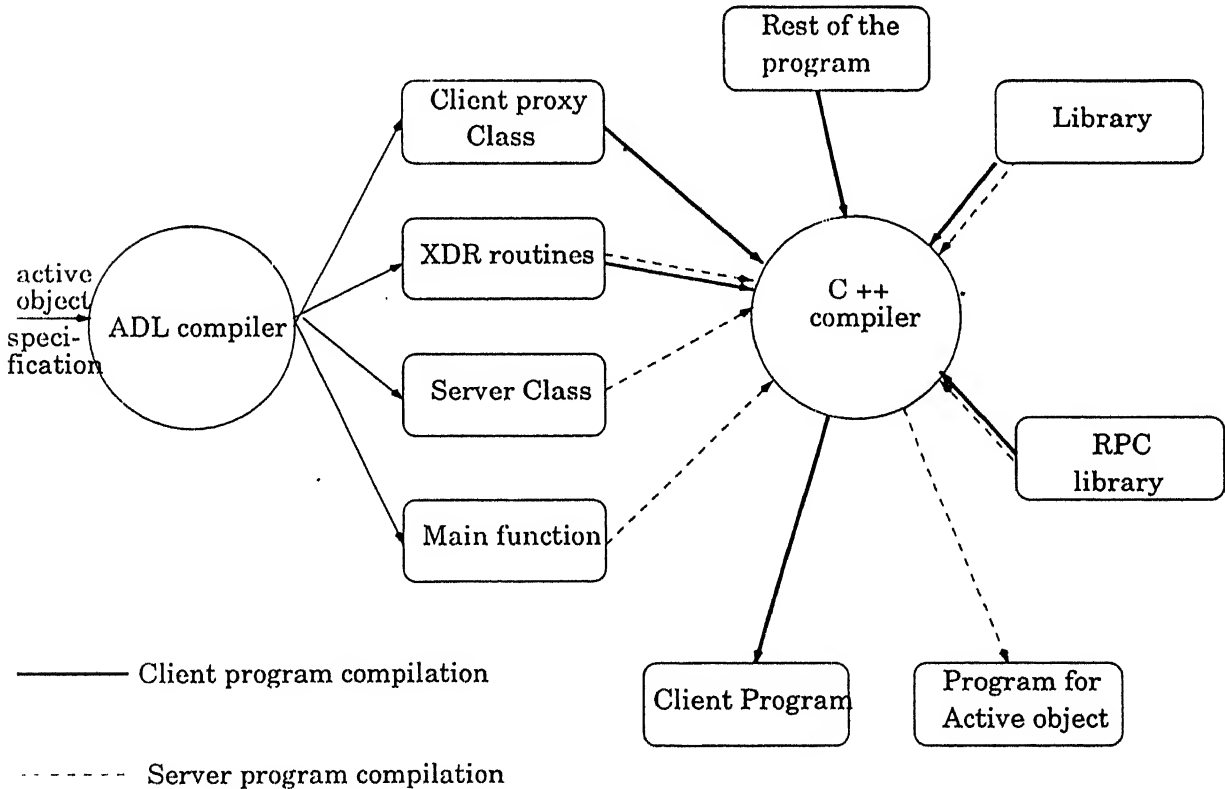


Figure 3: Using the ADL compiler

The generated client proxy class inherits from a predefined class called *active_proxy_root* by default. This class provides the data and functionality common to all client proxy classes. The declaration of the *active_proxy_root* class is shown in Figure 4.

```
class active_proxy_root
{
protected:
    HANDLE h;
    CLIENT *cl;
    int errno;
public:
    HANDLE get_handle();
    int get_errno();
    virtual HANDLE restrict(unsigned int bitmask);
};
```

Figure 4: *active_proxy_root* class declaration

The field *h* stores the handle of the active object. The member *cl* is the RPC style client handle that is created from the handle *h*. The *errno* field indicates any error that might have occurred in a remote method invocation. After every method invocation, the client checks this field to determine whether an error has occurred before actually interpreting the results. The *geterrno* method returns the value of the *errno* field. The *get_handle* method is needed in cases, when a client passes an active object to another active object. This is explained later. The *restrict* method is defined as a virtual method and needs to be defined in any proxy class that inherits from this class. The need for this method will become clear later. ADL compiler generates this method in the proxy classes generated.

The generated server class inherits from a predefined class called *active_root*, by default. This class provides the data and functionality common to all server classes generated. Its declaration is shown in Figure 5.

The *check* field is used for authentication purposes to be described later. The *code* field stores the value returned by a name server upon registration. This is

```

class active_root
{
protected:
    int check;
    int code;
public:
    int authenticate_handle(HANDLE h);
    int authenticate_request(HANDLE h,int procno);
    HANDLE restrict(unsigned int bitmask);
    virtual void main(char *objname,int fd);
    virtual void disptach(struct svc_req *rqstp,SVCXPRT *transp,
                          int handle_authenticated);
};

```

Figure 5: active_root class declaration

also explained later. The default authentication service is provided as the *authenticate_handle* method and the *authenticate_request* method of the *active_root* class and can be used by any active object. If an active object wants to use its own authentication method, it can override these methods. The *restrict* method is used to restrict the handle. The *main* method and the *dispatch* method are declared as virtual methods of the *active_root* class and should be defined in the derived classes. The ADL compiler generates these methods for each active class.

The generated client proxy class has the same name as the input active class. Two constructors are provided in the client proxy class. These two are needed for the two scenarios — one in which a proxy object needs to be created for an active object that is executing already; and the other in which both a proxy object and the corresponding active object need to be created. The first of these constructors takes an argument which is a handle. It assigns the handle in its private data *h*, converts this handle into RPC style handle and stores it in its private data *cl*. This constructor is used to create a client proxy object for an active object that is already executing and whose handle is already known. Usually, handle for an active object is obtained either by contacting the name server with the name of the object or by invoking the *get_handle* method of a proxy object of the active object. The second of

the constructors is used to actually create the active object. The constructor takes a pointer to a structure describing where the object should be created as a parameter. The code for this constructor contacts the object server on the remote machine to create the active object and obtains a handle for it. The proxy class has a method corresponding to each of the publicly accessible methods of the input active class with the same name and argument types. Upon invocation, this method forms a request message and sends it to the active object.

The server class has everything the input active class has in addition to the *main* and the *dispatch* methods. The *main* method creates the server side RPC transport handle, optionally registers the active object with the name server, returns its handle to the object server and calls a library function *acceptor* to wait for requests. When a request message arrives, this function invokes the *dispatch* method of the active class to dispatch the request message.

3.2 Object Handles

Each active object has a handle which acts as a capability for it. The structure of the handle is shown in Figure 6.

```
typedef struct handle
{
    unsigned int hostid;
    long port;
    unsigned int rights;
    int check;
}HANDLE;
```

Figure 6: HANDLE structure

The field *hostid* carries the internet address of the machine on which the process implementing the active object executes. The field *port* is the UDP port number on which the object waits for requests. The *rights* field indicates the access rights that the holder of the handle has on the active object. The *check* field is used for

authentication purposes as described later. Even though, the handle structure is not currently location and migration transparent, it is sufficient for the first version of the system and it simplifies the implementation. Later, this can be easily modified to support other features such as object migration etc.

3.2.1 Authentication

An active object is implemented as a process which uses RPC for its communication. The authentication mechanism that it uses is much more secure than the default *Unix* authentication scheme supported by RPC. The default authentication mechanism that active objects use is based on the rights and check fields of the handle. This scheme is based upon the authentication scheme in *Amoeba*[26] and is described below.

The authentication information in an object handle consists of a rights field and a check field. The rights field consists of 32 bits and a '1' in any bit denotes that the holder of the handle has the right to perform some particular operation on the object. The check field is used to prevent unauthorized accesses to the object and to ensure that the holder of a restricted capability for the object (i.e., one in which not all bits in the rights field are '1') cannot acquire more rights than it legitimately has.

When an active object is created, it generates a random number and stores it in a variable. The handle returned to the creator is an owner's capability for the object and thus has 1s in all bits of the rights field. The check field of the handle contains the generated random number.

When an active object gets a request message with a handle, it checks the rights field. If the client has full rights, the object compares the check field in the handle with the internally stored check field. If both of these are equal, the client is authenticated successfully.

A client can restrict the rights of a handle that it has and pass the restricted handle to some other client. To restrict a handle, the client passes the handle to the active object along with a bitmask indicating the rights that are needed. Upon getting this request, the active object authenticates the handle and creates a new handle. Then, it ANDs the rights field in the handle with the bitmask and stores the

result in the rights field of the new handle. Finally, it XORs the check field stored internally with the new rights field and applies a one-way function to this value. It stores the result in the check field of the new handle and returns the handle to the client.

When a client sends a request with a restricted handle, the object XORs the internally stored check field with the rights field in the handle and applies the same one-way function to it. The client is authenticated successfully, if the result is equal to the check field present in the handle.

It is obvious from the algorithm that a client that tries to add rights that it does not have will simply invalidate the handle. The client cannot get the original check field from the check field in a restricted handle, because the one-way function that is used is computationally hard to invert. The particular one-way function that we have used is a direct implementation of the algorithm proposed by Evans et al [10].

The authentication scheme authenticates the handle only. This is implemented as the *authenticate_handle* method of the *active_root* class and hence is available to all active classes. Once the handle has been authenticated, one needs to check whether the client is sufficiently privileged to invoke the requested method of the object. For this purpose, the bit corresponding to the requested method in the rights field of the handle has to be checked. Programmers are free to assign any bits to any methods. However, by default, the least significant bit of the rights field is interpreted as specifying the right for destroying the active object. The *authenticate_request* method of the *active_root* class takes a handle and a procedure number as arguments and determines whether the client has the right to invoke the destructor. Note that this can be implemented in the *active_root* class since the destructor method is always assigned the number 1. The programmer can assign meanings to more bits of the rights field by overriding the *authenticate_request* method of the *active_root* class. While doing this, it must be ensured that different classes in an inheritance hierarchy assign meanings to different sets of bits of the rights field. The *authenticate_handle* method can also be overridden if one wishes to use some other authentication scheme. However, we expect that while the *authenticate_request* method will be quite frequently overridden, the *authenticate_handle* method will be overridden very rarely.

The handle restriction facility is implemented by the *restrict* method of the *active_root* class. The *restrict* method of the proxy object invokes this method to restrict the handle.

3.3 Object Creation

Scheduling an active object at the time of creation is an important issue. By scheduling, we mean the selection of the machine on which the new object will be created. Explicit scheduling means that the machine on which the new object will be created is specified by the programmer and implicit scheduling means that the system determines the machine on which the new object is created. VASTU supports both implicit scheduling and explicit scheduling of objects. Currently, implicit scheduling means creating the object on the same machine as that of its creator. We support explicit scheduling of objects by allowing the programmer to specify a list of machines to be tried for object creation. As, we already discussed, the generated client proxy class has a constructor that takes an argument which is a pointer to a structure of type `OBJ_CR_PARAMS`. This structure is shown in Figure 7.

```
typedef struct object_creation_params
{
    char *machines[MAXMACHNAME];
    int n_machines;
    char *obj_name;
    char *pathname[MAXPATHNAME];
} OBJ_CR_PARAMS;
```

Figure 7: `OBJ_CR_PARAMS` structure

Using this structure, a client can specify a list of machines on which object creation has to be tried out. The *n_machines* field indicates the number of machines specified in the list. The *machines* field is an array of machine names. The *obj_name* field specifies a name for the object that should be registered with the name server. If a client does not want to register the active object with the name server, it can give a NULL value in this field. The *pathname* field is an array of pathnames of binary

executables corresponding to the active object on each of the machines specified. The binary executable file corresponding to an active class may not be submitted in the repository maintained by the object server. In this case, the programmer specifies the pathname of the binary executable file in the *pathname* field. If the binary executable file is submitted in the repository of an object server, the programmer can give a NULL value in the *pathname* field. In this case, the name of the class represents the name of the binary executable file in the repository.

A programmer who chooses to use explicit scheduling, needs to pass a pointer to an `OBJ_CR_PARAMS` structure containing the necessary information to the constructor. Otherwise, he passes a NULL pointer to use implicit scheduling. The constructor actually uses the *obj_create* function of our library to do the scheduling.

An object server maintains a repository of binary executable files corresponding to active classes. After compiling the server side code generated for an active class into a binary executable file, a programmer can submit it in this repository. The object server supports two services. The services are — creation of an active object given the name of the class; and giving load information upon request. The load information that is provided currently is the number of active objects present on the local machine. The load information is expected to be used for implicit scheduling but we do not use it currently. The object server is implemented as an RPC server and it currently uses the *Unix* authentication scheme supported by RPC for authenticating clients. This authentication can be easily broken as it is based on the *user id* and *group id* of the clients. However, DES authentication can be used if stricter security is required.

When a program implementing an active object begins execution, its *main* function gets called. This function creates an object of the active class and invokes the *main* method of the active object. The *main* function and the *main* method are generated by the ADL compiler. The *main* method creates a handle for the object. This handle needs to be returned to the object server that has created the active object. Unix pipes are used for this communication. The object server opens a *pipe* before forking the process that implements the active object. It passes the write file descriptor number of the pipe to this process as a command line argument. The *main* method writes the handle into the pipe using this file descriptor. The object server

reads the handle using the read file descriptor of the pipe and returns the handle to the client. Optionally, the client passes a name for the new object to the object server. The object server passes this name to the active object again as a command line argument. If this name is not NULL, the *main* method registers a restricted handle and the name with the name server.

As can be seen, this interface does not restrict the creation of active objects via only the object server. Active objects can also be created simply by executing the corresponding binary executable in the usual manner. Thus “daemon objects” can be easily created at boot time..

3.4 Invoking Remote Objects

A protocol needs to be followed between the proxy object and the corresponding active object. For each method invocation, the proxy object forms a request message and sends it to the active object. This message contains the parameters of the method, the handle of the object, a number indicating the method invoked and the name of the class. The need for the class name will become clear soon. Upon getting this message, the active object extracts the class name and the handle from the message, authenticates the handle, determines what method is invoked, authenticates the request, extracts parameters, invokes the method, forms a reply message containing the result and sends it back to the proxy object. The method number is passed in the procedure number field of the RPC message. The name of the class, the handle and the arguments are packed in that order into the data portion of the RPC message.

The arguments of the methods of the active object can be of any type supported by *rpcgen* such as integers, characters, floating point numbers, structures, arrays etc. In addition, active objects can be passed as arguments to methods of active objects. This is done by passing the handle for the active object to be passed. Thus active objects can only be passed by reference. The *get_handle* method of the client proxy class is used to obtain the handle of the active object being passed. On the server side, a client proxy object for that active object is created using the handle that is passed and a reference to the proxy object is actually passed to the method of the

called active object. The constructor of the proxy class with an argument of type `HANDLE` is used to create this proxy object. Passive objects can not be passed by reference, since they do not have network wide references such as handles. To support passing objects by value, the method code of the classes should be made available on the callee side. This is a difficult task and hence we do not support passing objects by value currently.

Whenever a request message for method invocation is received, the *dispatch* method of the object is invoked. It unpacks the class name and the handle from the message and authenticates the handle by invoking the *authenticate_handle* method. If the class name is same as the name of the active class, it authenticates the request by invoking the *authenticate_request* method, unpacks the arguments from the message based on the procedure number and invokes the appropriate method. Upon successful execution of the method, it packs the result into a message and sends it back to the client. The scenario is more complicated in the case of inheritance. This will be described in a later section.

3.5 Destroying Active Objects

In many situations, an active object needs to be destroyed once the proxy object corresponding to it is destroyed. Code for destroying active objects is generated by default. A *destructor* is added to the client proxy classes generated. This destructor sends an RPC message with procedure number 1 to the active object. When the *dispatch* method of the server class gets a request message with procedure number 1, it sends a reply immediately and calls the *exit* system call. This results in the call of the destructor of the active object automatically. The destructor of the active object removes its mapping from the name server. This is shown in Figure 8. A programmer may not want to destroy an active object that he has created, even if the corresponding proxy object is destroyed. Code for this can be generated by specifying an option to the ADL compiler.

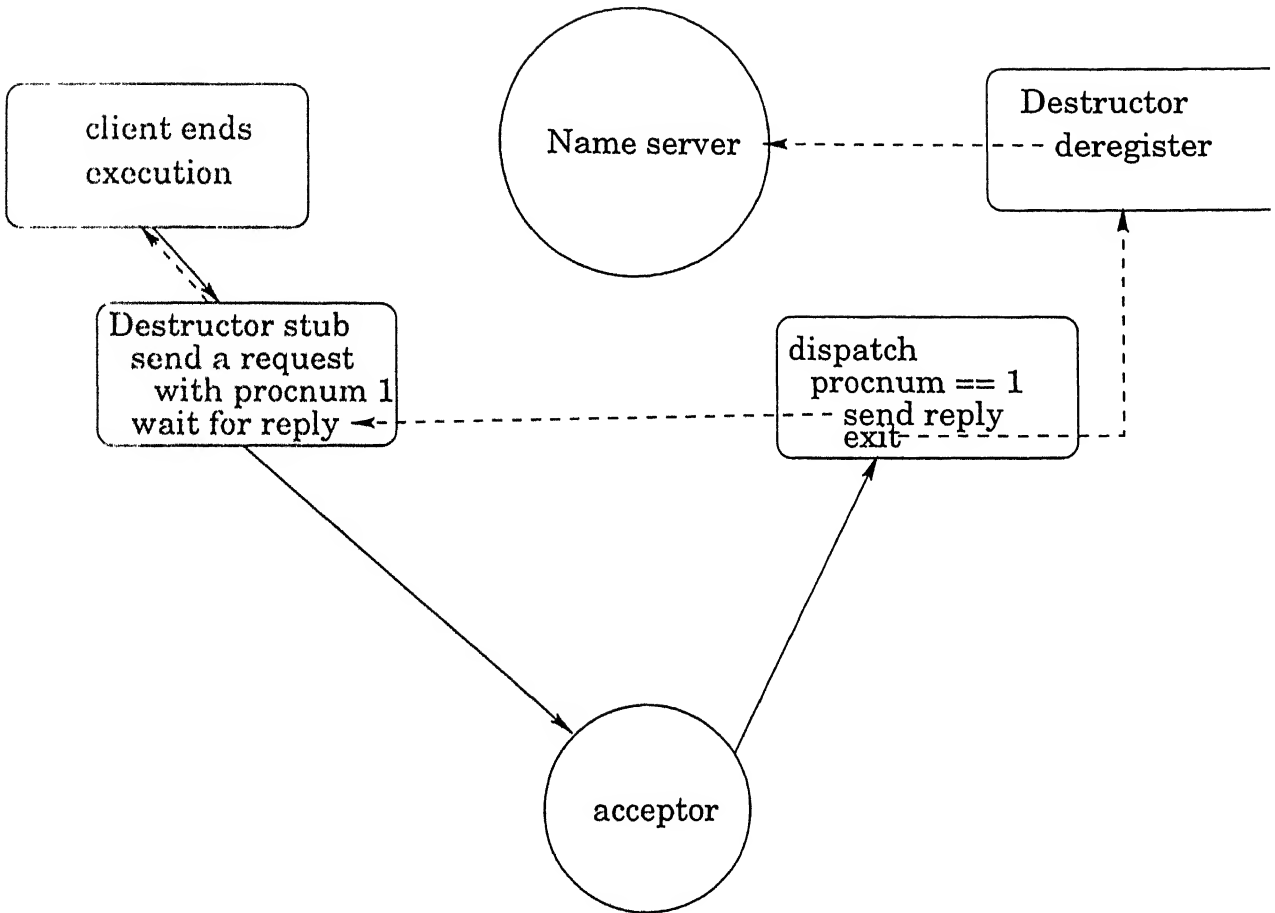


Figure 8: Destruction of an active object

3.6 Inheritance

Inheritance is a relationship between classes where one class(base class or super class) is the parent of another class(derived class or sub class). Inheritance is a natural way to model the world. It also provides for code and structural reuse. All methods and data members of a super class are available to all of its derived classes. This kind of reuse can occur within an application as well as across applications also. Inheritance is one of the key features of the object oriented programming paradigm. Hence, we need to support inheritance in VASTU also. VASTU allows active classes to inherit from other active classes. Passive classes can inherit from passive classes as usual.

Thus there are two class hierarchies – one in which all classes are active; and the other in which all classes are passive. Thus we do not support a class hierarchy in which some classes are active and others are passive. In our opinion, a programmer will not wish to do this, since the nature of passive objects and that of active objects are entirely different. Moreover, passive objects are created on the client side itself whereas only proxy objects of active objects are created on the client side. If a passive class inherits from an active class, in effect it is inheriting from the proxy class of the active class and hence an object of the passive class includes the state of an object of the proxy class only. Therefore, supporting a mixed inheritance hierarchy is a very difficult task.

To explain the issues involved in supporting inheritance, let us take a simple example. Consider the class hierarchy shown in Figure 9.

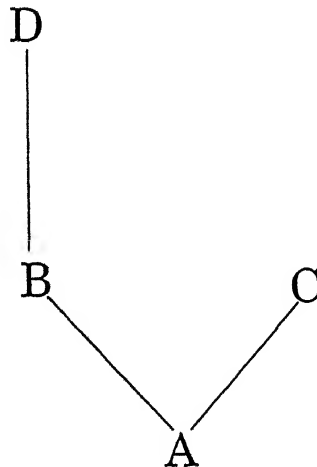


Figure 9: A simple inheritance hierarchy

In this example, the active class A inherits from two active classes B and C. The active class B in turn inherits from another active class D. In such a scenario, the programmer is required to write the declarations of these active classes in ADL in separate specification files. These files are then passed one by one through the ADL compiler. The ADL compiler generates a client-proxy class, server class, XDR routines and *main* function for each of the input active classes. The programmer compiles his program along with the client proxy classes and the XDR routines of

all active classes into a client program. The binary executable implementing class A clearly needs to contain code for all the four classes: A, B, C, and D. Thus to get the binary executable that implements objects of class A, the *main* function generated for the class A needs to be compiled along with all server classes and XDR routines generated.

3.6.1 Method Dispatching

As mentioned earlier, the *dispatch* method gets invoked when a request message is received by the active object. The *dispatch* method has to determine which method to invoke based on the request message that it gets. The client proxy object can send a number to indicate this if there is a single class. But, in the presence of inheritance where there are many classes each having its own methods, just a number cannot indicate the method to be invoked. Hence, we use the class name and a number indicating the method within the class to identify the method to be invoked on the active object.

Consider the example hierarchy shown in Figure 9. The same class hierarchy is preserved in the proxy classes as well as the server classes. Thus the generated proxy class for A inherits from the proxy classes for B and C and the proxy class for B inherits from the proxy class for D. Similarly on the server side, the server class for A inherits from the server classes for B and C and the server class for B inherits from the server class for D. Let us assume that the client invokes some method of class B on a proxy object of class A. The method code packs the class name (B), the handle, the arguments and the procedure number of the method into a message and sends it to the active object. Upon receiving the message, the active object invokes the *dispatch* method of Class A. The *dispatch* method of class A unpacks the class name and the handle from the message and authenticates the handle by invoking the *authenticate_handle* method. Then, it checks whether the class name is the name of one of its immediate super classes (B and C) or A itself. Since the class name is B (i.e., one of the immediate super classes of A), the *dispatch* method invokes the *dispatch* method of the class B with the rest of the message. The *dispatch* method of class B also checks whether the class name is the name of one of its immediate super classes

(D) or B itself. Since the class name is B, the dispatch method authenticates the request by invoking the *authenticate_request* method, unpacks the arguments based on the procedure number, invokes the required method, forms a reply message with the result and sends it back to the client.

Now, let us assume that the client invokes some method of class D on the same proxy object. The method code packs the class name (D), the handle, the arguments and the procedure number of the method into a message and sends it to the active object. Upon receiving the message, the active object invokes the dispatch method of Class A. The dispatch method of class A unpacks the class name and the handle from the message and authenticates the handle by invoking the *authenticate_handle* method. Since the class name is not one of the names of its immediate super classes (B and C) nor its own name (A), it invokes the dispatch methods of its immediate super classes one by one. If one of them succeeds in dispatching the request, the dispatch method of A returns. Hence, the dispatch method of class B gets invoked first. Since the class name (D) is the name of one of its immediate super classes, it invokes the dispatch method of D. The dispatch method of D dispatches the request as described earlier and returns control to its caller, the dispatch method of B. It returns control to the dispatch method of A which returns control back to the *acceptor* which once again waits for requests.

This kind of a recursive dispatching is necessary because while generating code for a class, the names of only that class and its immediate super classes are known. In the shown example, while the ADL compiler is generating code for class A, it does not know that the class A will also (indirectly) inherit from class D. Thus the generated dispatch method for class A cannot itself dispatch a request that invokes a method inherited from D.

3.7 Name Server

The name server maintains mappings between object names and handles. It is implemented as an active VASTU object and has public methods to register a mapping, deregister a mapping and to lookup a name. Thus, it can be used to obtain the

handle for any publicly accessible object. But, a client can contact the name server only if it knows the handle of the name server. This problem is solved by making the handle of the name server well-known. Thus, the name server listens to a fixed UDP port on each machine. A library routine called *get_namehandle* takes a machine name as argument and returns the handle for the name server on that machine. The name server overrides the default *authenticate_handle* method by a method that always reports that the handle was valid. The default *authenticate_request* method is also overridden by a method that, regardless of the rights field, allows all methods to be invoked except the destructor method. Hence, the values that the *get_namehandle* routine puts in the rights field and check field of the handle are irrelevant.

The name server needs to prevent a malicious client from deregistering a mapping which was registered by some other client. This can be ensured by following a scheme similar to the authentication scheme. Whenever a request for registration arrives, the name server generates a random number, stores it along with the mapping and returns it to the client. Later, when the client wants to deregister a mapping, it needs to provide this number along with the name of the object.

Some active object might have registered itself with the name server and might have terminated without removing its mapping from the name server. Since such invalid mappings present with the name server lead to wasted effort, they should be removed from the list of mappings maintained by the name server. The *main* method of the name server class creates an additional thread that does this periodically. To check the validity of a mapping, it invokes the method with procedure number 0 on the active object. Each active object returns a dummy reply message whenever it receives a method invocation request message with procedure number 0. So, all active objects which do not respond to this invocation request are considered to be dead and their mappings are removed from the list.

3.8 A typical Execution Scenario

In Figure 10, a typical execution scenario is shown. A client begins execution, creates a proxy object and passes a pointer to OBJ_CR_PARAMS structure to the constructor.

The constructor calls *obj_create* library routine which contacts an object server on the machine specified in the structure. That object server executes the binary executable corresponding to the active class. The *main* function of the process implementing the active object invokes the *main* method of the active object. This method creates an RPC server handle and returns a handle for the object to the object server. The object server forwards it to the client. The constructor of the proxy object then translates this handle to the RPC style handle by calling a library routine *translate*. In the meanwhile, the *main* method of the active object registers a restricted version of its handle along with a name passed to it by the object server with the name server and calls the *acceptor* routine of the library to wait for requests.

Later, when the client invokes a method of the proxy object, the method code packs its class name, the handle and arguments into a message and sends it to the active object. The *acceptor* routine reads it and invokes the *dispatch* method of the object. It unpacks the class name and the handle from the message, authenticates the handle by invoking the *authenticate_handle* method and checks whether the class name is same as the name of the active class. Then, it authenticates the request by invoking the *authenticate_request* method, unpacks the arguments based on the procedure number present in the message and invokes the method of the active object. On successful completion of the method, the dispatch method forms a reply message containing the result and sends it to the client proxy object's method. The client proxy object's method unpacks the result from the reply message and returns it to the caller. Meanwhile, on the server side, the *dispatch* method of the active object returns control to the *acceptor* routine. The *acceptor* routine once again waits for requests.

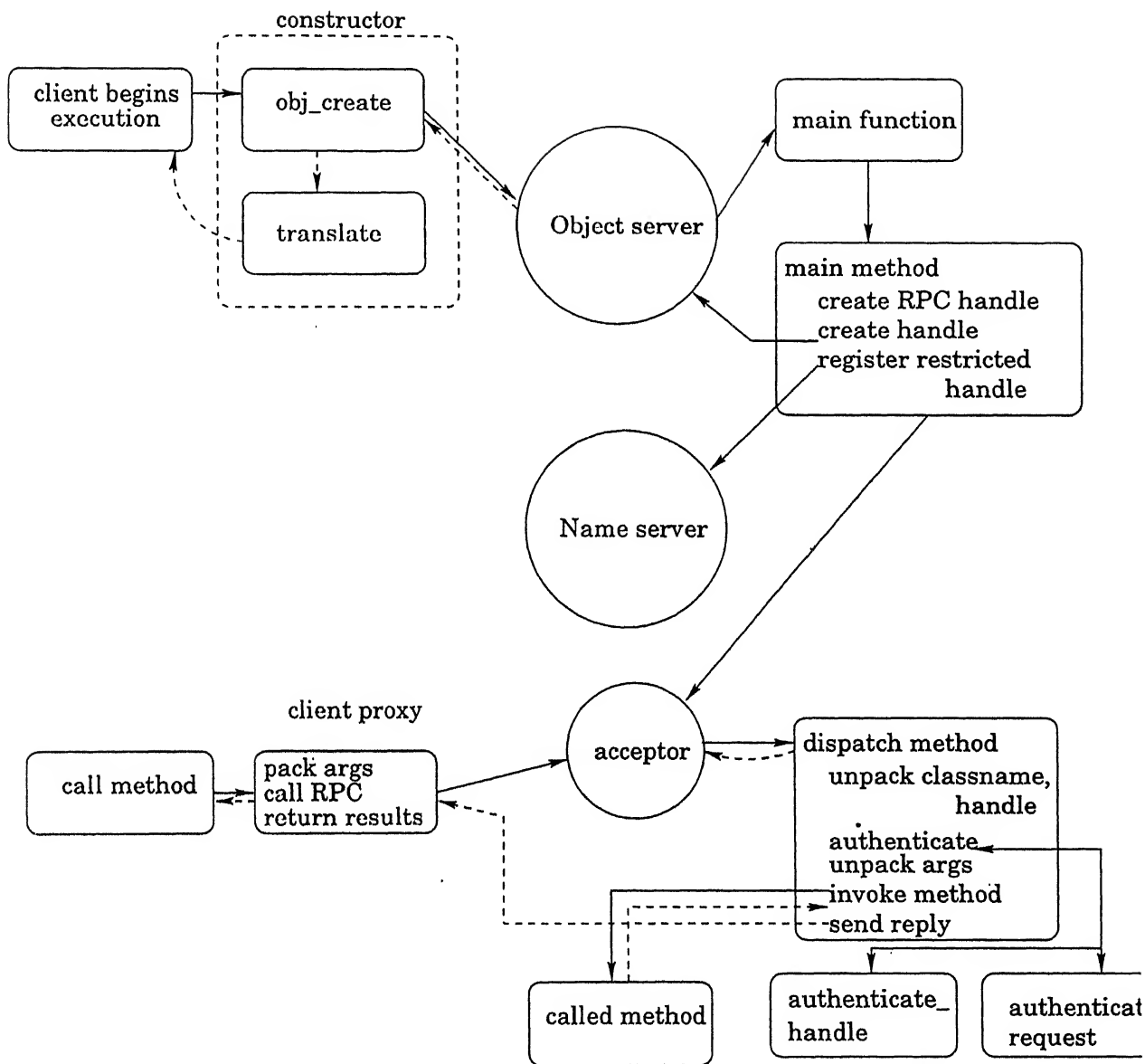


Figure 10: An execution scenario

Chapter 4

An Example Application

In the previous couple of chapters, we described the design and implementation of VASTU. An example will make it clear as how to develop distributed object oriented programs in VASTU. The example program is a client-server kind of program, in which an active object acts as the server.

4.1 Description of the Example

The problem that we have taken in this example is the implementation of a print server similar to the printer daemon *lpd*. The server controls the printer queue of a printer. It provides services for adding an entry into the queue, removing an entry from the queue and getting the status of the queue. The server has to continuously schedule the jobs present in the queue one by one on the printer. It is implemented as an active object which has the printer queue as its state and which provides methods (services) to submit a job, cancel a job and to get the status. The active object does the scheduling of jobs as a background activity. Clients which want to access the services of the print server are provided with the declaration of the client-proxy class of the print server class. Therefore, they create proxy objects for the print server object and access the services by invoking the methods of the proxy object.

4.2 The ADL Specification of the Print server class

The ADL specification for the print server class is shown in Figure 11.

```
#include <stdio.h>
#define MAXNAMESZ 50
%%
typedef struct Qelement *Qptr;
struct Qelement
{
    char filename[MAXNAMESZ];
    char owner[MAXNAMESZ];
    int size;
    int jobno;
    Qptr next;
};
%%
active class printserver
{
    private:
        Qptr head;
        Qptr tail;
    public :
        int submit(char filename[MAXNAMESZ],char owner[MAXNAMESZ],int size);
        int cancel(int jobno);
        Qptr status();
};
```

Figure 11: ADL specification of the print server class

The printer queue is implemented as a linked list whose nodes contain the information about the jobs. Each node in the list has the name of the file to be printed, its size, the login name of the person who has submitted the job and the job number. Each job is given a job number after it is successfully submitted into the queue. The state of the print server object has a pointer to the first node of the queue(*head*) and a pointer to the last node of the queue(*tail*). As shown in Figure 11, the print server class provides three methods *submit*, *cancel* and *status* to do the three services mentioned earlier.

4.3 Generating Code from the ADL Specification

Two optional switches can be enabled when we generate code using the ADL compiler. First switch specifies that concurrent dispatching scheme is needed and the second switch specifies that code for destructor should not be generated in the client proxy class generated. A sequential dispatching scheme is sufficient for the print server object, since it is meant for explanatory purposes. Therefore, the first switch should not be enabled. The print server object should not be destroyed once the proxy object that has created is destroyed. Therefore, the second switch needs to be enabled. Hence, the ADL specification of the print server class is passed through the ADL compiler with the second switch(specifying that the destructor should not be generated in the proxy class) enabled. The ADL compiler generates the files — *printserver.cnt.h*, *printserver_serv.h*, *printserver_serv.cpp*, *printserver_main.cpp* and *printserver_xdr.c*. The file *printserver.cnt.h* contains the declaration of the client-proxy class and the definition of its methods. The file *printserver_serv.h* contains the declaration of the server class. The definition of the *dispatch* method and the *main* method are present in the file *printserver_serv.cpp*. The *printserver_main.cpp* file contains the code of the server side *main* function. The XDR routines are present in the file *printserver_xdr.c*.

4.4 Building the Print Server from the Generated Code

The declaration of the print server class present in the *printserver_serv.h* file is modified so that it includes the declaration of a new method called *schedule* also. The *schedule* method schedules the job queue. It removes the jobs one by one from the queue and simulates printing by waiting for some time duration proportional to the size of the job before proceeding with the next job. The definition of the *submit* method, *cancel* method, *status* method and the *schedule* method are written in a separate file *printd.cpp*. The generated *main* method code is modified so that it creates a new thread to schedule the jobs before calling the *acceptor* library routine to wait

for requests. The *schedule* method of the print server has to synchronize with other methods of the print server, because it is concurrently accessing the state of the print server object along with one of those methods. A mutual exclusion lock (mutex) of type *pthread_mutex_t* supported by the *pthread* library is used for this purpose. The mutex is initialised in the *main* method. The modifications done to the *main* method are shown in Figure 12. The newly created thread in the *main* method actually starts

```
..
..
#include <pthread.h>

..
..
extern printserver obj;
..
..
pthread_mutex_t m;
pthread_t t;

void *schedule(void *x)
{
    obj.schedule();
}

void printserver::main(char *name,int fd)
{
    ..
    ..
    pthread_mutex_init(&m, pthread_mutexattr_default);
    pthread_create(&t, pthread_attr_default,
        (pthread_startroutine_t) :: schedule, (pthread_addr_t) NULL);
    acceptor();
}
```

Figure 12: Modification of the generated main method

with a function called *schedule* which directly invokes the *schedule* method. This is due to the limitation of the *pthread_create* function of the *pthread* library that it needs a function as the starting routine of the new thread.

Any method of the print server class which accesses the state of the object, locks the mutex that is initialized in the *main* method before accessing the state. The method unlocks the mutex immediately after it has accessed the state. The portion of the code which does this in the *schedule* method and the *submit* method is shown in Figure 13. The same is done by the *cancel method* and the *status* method also.

```
int printserver::submit(char filename[MAXNAMESZ],
                        char owner[MAXNAMESZ],int size)
{
    /* create a new node and copy the arguments
       into the new node*/
    pthread_mutex_lock(&m);
    /* add the node at the end of the queue*/
    pthread_mutex_unlock(&m);
    return 0;
}

void printserver::schedule()
{
    while(1)
    {
        pthread_mutex_lock(&m);
        /* remove the first element from the queue*/
        pthread_mutex_unlock(&m);
        /* simulate printing*/
    }
}
```

Figure 13: Synchronisation between the schedule method and the status method

The files *printserver_main.cpp*, *printserver_serv.cpp*, *printserver_xdr.c* and *printd.cpp* are compiled into a separate binary executable file *printserver*. The binary executable can be executed directly from the shell or through the object server. Hereafter, any other client can access this print server object by obtaining a handle from the name server and by creating a proxy object of the print server with that handle.

Chapter 5

Conclusion

In this report, we have described the design and implementation of VASTU, a distributed object oriented programming environment. VASTU supports both passive objects and active objects. Objects are created, used and destroyed in a transparent manner. There is a separate specification language(ADL) in which the programmer writes the declaration of an active class. When the ADL specification of an active class is given as input, the ADL compiler generates code which the programmer uses to transparently access objects. The details of the code that is generated and how the programmer uses the code are explained in the report.

VASTU uses a naming scheme that is based on capabilities and name server. Each active object has a *handle* that acts as the capability for it. The name server is implemented as an active object whose handle is well known. An object server runs on every machine in the system and is responsible for creating active objects on that machine. VASTU supports a default authentication scheme based on handles and control procedures. This scheme is highly flexible and programmers can tailor it easily for each active object.

VASTU supports development of distributed programs using an existing object oriented programming language, namely C++. As we saw earlier, VASTU has a separate specification language for specifying distributed classes. The language is syntactically very similar to C++ and hence there is very little difficulty for programmers in using VASTU.

We saw earlier that in each feature of a distributed object oriented programming system, there are many alternative schemes that can be supported. VASTU support for a basic set of those features currently. But, for each of those features, all alternative schemes are supported in VASTU in a flexible manner. The programmer is free to choose the alternatives based on his need and tailor them. This is not the case in many existing systems which either support these features in a rigid manner or do not support them at all. A comparison of VASTU with some existing systems will make this clear. VASTU supports the notion of passive objects and the notion of active objects unlike Argus and Eden which support active objects alone and Sloop and Aeolus which support passive objects alone. Unlike CORBA, VASTU supports creation of objects transparently. Inheritance is supported in VASTU whereas systems like Emerald, Argus and Eden do not support it. VASTU has a security scheme that is highly flexible unlike the rigid security schemes of CORBA and Eden. The security scheme is based on capabilities and control procedures. Emerald, Sloop and Argus do not provide a security scheme at all. VASTU allows the programmer to choose between a sequential dispatching scheme and a concurrent dispatching scheme. The ADL compiler of the system generates code in C++ and the programs written in VASTU can run on any machine running the Unix operating system and supporting RPC. But, programs written in systems such as Emerald, Aeolus and Argus are not portable.

The system that we have described here is only the first version of VASTU. Currently VASTU does not support features such as implicit scheduling of objects, passing objects by value to methods of active objects, distributed naming service and reliability. But, it is flexible enough so that these features can be easily accommodated in future.

Bibliography

- [1] ALMES, G., BLACK, A., LAZOWSKA, E., AND NOE, J. The Eden system: A technical review. *IEEE Transactions on Software Engineering* 11, 1 (Jan 1985).
- [2] BAKER, S. *CORBA Distributed Objects-Using Orbix*. Addison-Wesley Longman Limited, Essex, United Kingdom, 1997.
- [3] BAL, H. E., STEINER, J. G., AND TANENBAUM, A. S. Programming languages for distributed computing systems. *ACM Computing Surveys* 21, 3 (Sep 1989), 304–305.
- [4] BENER, M., GEINS, K., HEUSER, L., MUHLBARSO, M., AND SCHILL, A. Distributed systems, osf dce, and beyond. In *Proceedings of the international DCE workshop, Oct 1993, Lecture Notes in Computer Science* (1993).
- [5] BIRRELL, A., AND NELSON, B. Implementaing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2 (Feb 1984), 39–59.
- [6] CHIN, R. S., AND CHANSON, S. T. Distributed object-based programming systems. *ACM Computing Surveys* 23, 1 (Mar 1991).
- [7] CLARK, K., AND GREGORY, S. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems* 8, 1 (Jan 1986), 1–49.
- [8] DASGUPTA, P., AND LEBLANC, R. The Clouds distributed operating system. *IEEE Computer* 24, 11 (Nov 1991), 34–44.
- [9] DIGITAL EQUIPMENTS CORPORATION, MASSACHUSETTES. *Guide to DEC threads*, Jul 1994. Part II: POSIX 1003.4a(pthread) Reference.

- [10] EVANS, A., KANTROWITS, W., AND WEISS. A user authentication scheme not requiring secrecy in the computer. *Communications of the ACM* 17, 8 (Aug 1974).
- [11] GELERTER, D. Genetive communication in Linda. *ACM Transactions on Programming Languages and Systems* 7, 1 (Jan 1985), 80-112.
- [12] GOLDBERG, A., AND ROBSON., D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass., 1983.
- [13] GRAY, J. Notes on database operating systems. *Lecture Notes in Computer Science* (1978), 393-481.
- [14] HPF compiler. http://www.csg.lcs.mit.edu:8001/mcrctr/mcrc-annual-report-93/section3_4.html.
- [15] HUDAK, P. Exploring parafunctional programming: Separating the what from the how. *IEEE Software* 5, 1 (Jan 1988), 54-61.
- [16] LISKOV, B. Distributed programming in Argus. *Communications of the ACM* 31, 3 (Mar 1988), 300-312.
- [17] LUCCO, S. E. Parallel programming in a virtual object space. In *OOPSLA'87 Proceedings* (Oct 1987), pp. 26-34.
- [18] MCKENDRY, M. S., AND HERLIHY, M. Time driven orphan elimination. In *IEEE Sixth Symposium on Reliability in Distributed Software and Data Base Systems* (Jan 1986), pp. 42-48.
- [19] RAJ, R. K. Emerald: A general-purpose programming language. *Software-Practice and Experience* 21, 1 (Jan 1991), 91-118.
- [20] RICHARD STEVENS, W. *Unix Network Programming*. Prentice Hall of India Ltd., New Delhi, 1996.
- [21] Split C compiler. <http://www.cs.berkeley.edu/projects/parallel/castle/split.c/>.
- [22] STROUSTRUP, B. *The C++ Programming Language*. Addison-Wesley, Massachusetts, 1995.

- [23] SUN MICROSYSTEMS INC. *Network Programming (Release 4.0)*, May 1988.
- [24] SUN MICROSYSTEMS INC. *RPC: Remote Procedure Call Protocol specification*, June 1988. RFC 1057.
- [25] SUN MICROSYSTEMS INC. *XDR: eXternal Data Representation Protocol specification*, June 1988. RFC 1014.
- [26] TANENBAUM, A. S. *Distributed Operating Systems (pp.376-430)*. Prentice Hall, New Jersey, 1995.
- [27] VINOSKI, S. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications* 14, 2 (Feb 1997).
- [28] ZPL compiler. <http://www.cs.washington.edu/research/zpl/>.

Appendix A

Active Class Description Language Grammar

```
specification: SECTION1 SECTION2 section3
              ;
section3      :                               /*No section3*/
              | ACTIVE CLASS ID inheritance '{' body '}' ';'
              ;
inheritance   :                               /*No inheritance*/
              | INH
              ;
body          :                               /*No body*/
              | body_part body
              ;
body_part     : visibility dec_list
              ;
visibility    :                               /*No visibility=>private*/
              | PRIVATE ':'
              | PUBLIC ':'
              | PROTECTED ':'
              ;
```

```

dec_list      : fun_data dec_list_fol
               ;

dec_list_fol  :                               /*No follower for declaration list*,
               | fun_data dec_list_fol
               ;

fun_data      : type ID fun_data_fol
               ;

fun_data_fol  : ';'                               /*Data member*/
               | '(' arg_list_fol                /*Function or method*/
               ;

arg_list_fol  : ')' fun_body
               | arg_list ')' fun_body
               ;

arg_list      : type id arg_follower
               ;

arg_follower  :                               /*prev arg was last|no argument*/
               | ',' type id arg_follower
               ;

type          : TYPE                               /*Ordinary type*/
               | VIRTUAL TYPE                     /*Virtual function type*/
               | ACTIVE TYPE                       /*Active class type*/
               ;

id            : ID                               /*normal identifier*/
               | ID '[' NUM ']'                  /*array*/
               | ID '[' ID ']'                   /*array*/
               ;

fun_body      : ';'                               /*No body. Just a prototype*/
               | '=' '0' ';'                     /*pure virtual function*/
               | FUNBODY                          /*Body is present*/
               ;

```